APX ATARI® PROGRAM EXCHANGE

Jim Dunion

DUNION'S DEBUGGING TOOL (DDT)

A debugging tool for use with the ATARI Macro Assembler™

Diskette: 16K (APX-20150)

User-Written Software for ATARI Home Computers

Jim Dunion

DUNION'S DEBUGGING TOOL (DDT)

A debugging tool for use with the ATARI Macro Assembler™

Diskette: 16K (APX-20150)

DDT
(Dunion's Debugging Tool)

by

Jim Dunion

Distributed By

The ATARI Program Exchange
P.O. Box 3705
Santa Clara, CA 95055

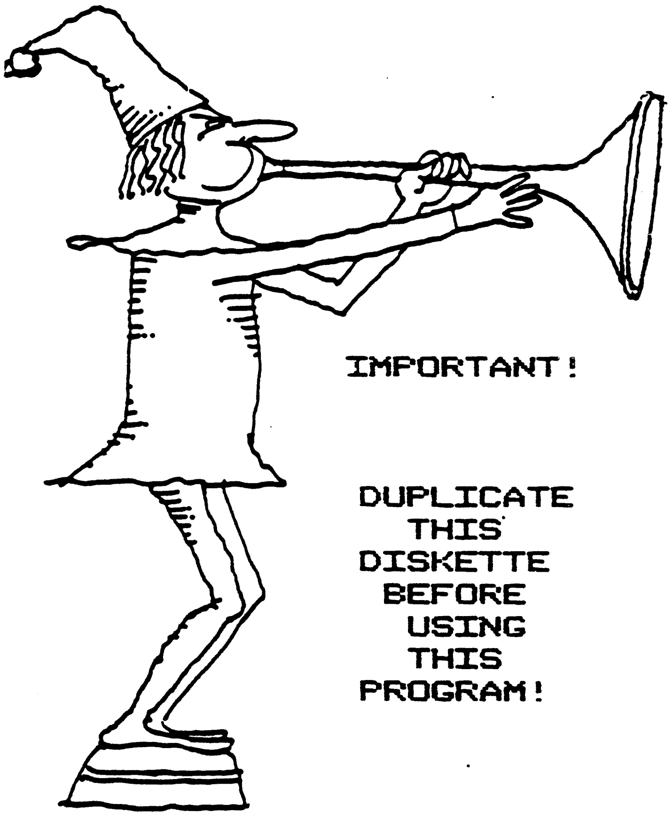
To request an APX Product Catalog, write to the address above, or call toll-free:

800/538-1862 (outside California)
800/672-1850 (within California)

Or call our Sales number, 408/727-5603

Trademarks of Atari

Printed in U.S.A.



This APX diskette is unnotched to protect the software against accidental erasure. However, this protection also prevents a program from storing information on the diskette. The program you've purchased involves storing information. Therefore, before you can use the program, you must duplicate the contents of the diskette onto a notched diskette that doesn't have a write-protect tab covering the notch.

To duplicate the diskette, call the Disk Operating System (DOS) menu and select option J, Duplicate Disk. You can use this option with a single disk drive by manually swapping source (the APX diskette) and destination (a notched diskette) until the duplication process is complete. You can also use this option with multiple disk drive systems by inserting source and destination diskettes in two separate drives and letting the duplication process proceed automatically. (Note. This option copies sector by sector. Therefore, when the duplication is complete, any files previously stored on the destination diskette will have been destroyed.)

CONTENTS

INTRODUCTION

THE ART OF CREATIVE PROGRAM DEBUGGING

In the simplest terms, a computer program is a sequence of operations that cause the computer to do something. Programming is merely the task of preparing instructions for the computer to execute. That seems simple enough, yet programming maintains an aura of mystery about it, so that even the most grizzled veteran of the computer wars approaches programming tasks with deference and hesitation. Try to get a programmer to commit to when a program will be ready, and you'll see what I mean. Even harder is getting a programmer to keep the few commitments made, because experience has burned many a hotshot programmer who promised the world but delivered the Bronx. DUNION's First Law is that things are never as simple as you thought they were going to be. In programming this means that programs invariably take longer to program than they should--to get them working, anyway. What is so difficult about programming computers? The answer is mainly that as humans, we aren't used to thinking as precisely as one has to in programming computers. Even the most rational human finds his thought processes tempered by emotion, intuition, and insight so that the resulting melange is greater than any completely rational, logical (linear if you will) sequence could be. Unfortunately, computers don't work that way (yet); they must be instructed in precise terms. And with machines that carry out some half million instructions every second, you don't have to be too far wrong in a program before disaster strikes. By the time you notice that anything wrong is happening, it has already happened. The problem is going from the conceptual to the concrete, from taking an idea and turning it into a program.

Who hasn't thought of the better software mousetrap? Somehow it's easier thinking up ideas than it is programming them. But it doesn't have to be that way. As any craftsman will tell you, much of the problem lies in not having the right tools. Programming as a human enterprise is somewhere between an art and a science, and no one is sure exactly where the line is drawn. Programming is hindered by inadequate software tools, but much of the problem is the attitude and approach towards the act of programming itself. I consider myself to be as much an artist as a technician; each new program is a new work of art. Not only does the program have to work correctly, but it also has to look right and feel right. The computer is an instrument of imagination, a paint brush beyond comparison, a pencil filled with millions of untold tales--the ultimate instrument, waiting for the performer to bring it to life. I call this attitude the art of creative computer programming.

I hate to admit it, but somehow mistakes work their way into my programs, particulary if I'm trying something new--working with a real-time system, perhaps, with color graphics and sound. A system like the ATARI 400/800 Computer is a good example. To reach the full potential of this system, we sometimes have to use assembly language programming. At this level of intimacy with the computer, every tiny mistake is magnified a thousandfold, and finding those mistakes is tough. It could be a syntax error, a semantic error, a timing error, a hardware error, an alpha ray zap, ..., the list goes on and on. As Piet Hein said in one of his Grooks, "Problems worthy of attack, prove their worth by hitting back".

Friends, I'm tired of this. What we need is something that can let us do a little bit better job of debugging, some creative debugging. What we need is something like -- DUNION'S DEBUGGING TOOL!

REQUIRED ACCESSORIES

16K RAM
ATARI 810 Disk Drive
ATARI Macro Assembler™ and Program-Text Editor™ (CX8121)

OPTIONAL ACCESSORIES

ATARI BASIC Language Cartrige (for examples)

CONTACTING THE AUTHOR

Users wishing to contact the author may write to him at:

1196 Borregas
P.O. Box 427
Sunnyvale, CA 94086

DDT USER'S GUIDE

DDT'S DESIGN PHILOSOPHY

The ATARI 400/800 Computer has features that set it apart from other current personal computers. Unfortunately, trying to get to these features from BASIC or PILOT is frustrating. In many instances, the only answer is to write at least a portion of the program in assembly language. That still wouldn't be too bad if we had decent assembly language development tools, but until very recently we didn't. That situation changed recently with the release of the ATARI Macro Assembler, a very powerful programming tool. However, considering that assembly language programs are wont to be bug-ridden at first (i.e., full of programming mistakes), the Macro Assembler emphasizes a serious need. Namely, what do we do about debugging assembly language programs? The ideal solution, of course, would be to have access to something like a logic analyzer or other type of hardware development system. Most of us don't, however. Then there's always the ATARI Assembler Editor cartridge. Without belaboring the point, this didn't strike me as a good idea. So what to do? The answer seemed to be to develop a debugging tool specifically designed for use with the Macro Assembler. Thus was born DDT.

DDT is a flexible, extensible, source language debugging tool. That means you would generally assemble DDT along with your source code as a sort of parasite. You can attach DDT to whatever is running inside the ATARI 400/800 Computer System. These attachments or "hooks" let DDT coexist with your test program. This flexibility is useful in a couple of ways. First, it lets you decide where DDT should reside in memory, which may vary, depending on exactly what is being debugged. Second, it lets you use the assembler to set up several of DDT's features. Note, however, that DDT is flexible enough that you don't have to assemble it with your program each time. The examples included on the DDT diskette will give you an idea of some ways to set up DDT (i.e., attach DDT to a program).

Most program bugs arise from assumptions (either explicit or implicit) that prove not to be true. If this is the case, a debugging tool that forces you to ask to see various locations, registers, breakpoints, and so on, misses a crucial point. Many times you have no idea at first what is causing a problem. The central idea in DDT is to place as much information as possible on the screen and then let your visual pattern recognition system (i.e., your eyes and right side of the brain) go to work. In short, let the computer do what it does best and let human programmers do what they do best.

A consequence of this approach is that DDT centers around control of its display screen. This control is coupled with the ability easily to change and monitor the internal state of the machine so that you can get a much clearer picture of exactly what's going on inside the system at any instant. Most of the time, correcting a program bug is easy; finding it is the trick. That's where DDT comes in.

The next section describes each DDT feature. Following that is a section explaining how to get started using DDT and describing the examples. Finally a technical appendix contains more information on how some of DDT's features are implemented. Read quickly through the entire manual to get an overview of DDT, and then go back and read each section more carefully. Finally, before you begin experimenting, take a blank diskette, format it, and write new DOS files on it. Then copy whichever of the source or object code modules you're interested in. If you want to experiment with one of the object code

modules, rename it as AUTORUN.SYS. Then all that you have to do is turn the machine off and back on to load and initialize the code automatically. What could be easier?

Happy hunting!

THE DDT SCREEN DISPLAY

The DDT screen display is how DDT shows you the internal state of the machine. The screen is divided into several display areas, each of which shows a different aspect of what is going on inside the computer at that instant.

The display areas are called :

REGISTER DISPLAY - a display of the current contents of 6502 registers
DISPLAY WINDOW - a window into memory
STACK DISPLAY - a display of the top 15 items on the system stack
MINI SYMBOL TABLE - a table of names and values of current symbols
BREAKPOINT TABLE - a table of the settings of breakpoint registers
COMMAND WINDOW - a window showing keyboard commands entered

The following sections describe each display area. Figure 1 is an example of a typical DDT display screen.

LOC	VAL	INSTRUCTION	STK	VAR.	VALUE
1E2F	20		81	LOMEM	33E0
1E30	75		96	MEMTOP	34E4
1E31	20		23	SYMB1	B8
1E32	00	PLA	45	LABEL1	A9
1E33	AA	TAX	76	LABEL2	00
1E34	68	PLA	97		
1E35	A8	TAY			
1E36	68	PLA			
1E37	40	RTI			
1E38	4C	JMP LABEL1			
1E39	BE				
1E3A	FF				
1E3B	A9	LDA #53			
1E3C	53				
1E3D	30	BMI 2 1E41			

BKP1	BKP2	BKP3	BKP4	BKP5	BKP6	TRP1	TRP2
ABCD	0000	0000	0000	0000	0000	ABC6	0000

PC	ACC	NV BDIZC	X	Y	SP	COMMAND
1E32	1E	00110100	00	12	F9	S 302122

Figure 1 Typical Screen Display

REGISTER DISPLAY

The lower part of the display screen displays the current contents of the 6502 processor registers. Whenever DDT is entered, the contents of its registers are copied into register shadows, which are then displayed. These shadows are used to restore the 6502 registers before control is released back to the program being tested.

These registers have their contents shown in hexadecimal notation:

```
PC  = Program counter, a two-byte value
ACC = Accumulator
X   = X index register
Y   = Y index register
SP  = Stack pointer
```

The Processor status register (NV BDI ZC) is shown in binary form, where

```
N = Negative flag
V = Overflow flag
B = BRK instruction flag
D = Decimal mode flag
I = Interrupt disable flag
Z = Zero flag
C = Carry bit
```

DISPLAY WINDOW

The display window forms a window into the system memory address space. This window is located in the upper left-hand portion of the display screen, and occupies more than a quarter of the screen. The window is set upon entry to DDT, or may be moved by single stepping, and by either the "E", the up-arrow, or the down-arrow command.

The window may be thought of as having one of three possible filters in front of it. You can change these filters by using the "W" command (see Command Interpreter section). The first filter, which is set upon initial entry to DDT, is an opaque filter. It has a summary of operating instructions written on it. With this filter in place, many commands will appear to do nothing.

The second filter is a disassembly filter. A greater than sign (>) points to what is called the current position. When DDT is entered, this will correspond to the value in the PC. The current position may be modified by the "E", up-arrow, or down-arrow command.

The third filter is a hexadecimal filter. The window shows the hexadecimal value and ATASCII representation of up to 48 memory locations. Again, the > sign indicates the current position.

There are always three bytes shown above the current position. These are shown in hexadecimal form.

In the disassembly display, each line from the current position down is shown in a similar format: first the hexadecimal address of a location, then its contents, and then a disassembly readout. Standard 6502 mnemonics are used, with conventional address mode indications.

Several features have been added to aid debugging. A mnemonic shown in inverse video indicates that a breakpoint has been set at that location. In fact, if you look at the actual contents of that location, it will be a 0. A BRK instruction in inverse video means that particular BRK instruction was not placed there by DDT. This would occur, for instance, in looking at memory that is all zeros.

Second, if the instruction is one of the branch instructions, an up or down arrow is added to the disassembly display to indicate the direction of the conditional branch. The computed address of the conditional branch location is also shown.

Finally, if the address portion of an instruction contains an address defined in the minisymbol table, the symbol name will be shown rather than the hexadecimal value. The symbol feature may be used to locate references to a symbol in the code, or simply as labels to make the disassembly listing more readable.

If the hexadecimal filter is in place, each line after the current position line will start on an even four-byte boundary. This means the current position line can have 1 - 4 values on it. The current position line values will always be left justified.

STACK DISPLAY

The middle portion of the upper display screen shows the top locations in the system stack. If the stack pointer is set at $E0 or higher (i.e., there are less that 15 entries in the stack), then only those values currently in the stack will be shown. The display is a top down representation. If more than 15 entries are in the stack, then only the top 15 are shown.

Examples

SP=$FF	SP=$FE	SP=$FD	SP=$F0	SP=$EF
	B9	B9	B9	A8
		A8	A8	A7
			A7	A6
			A6	A5
			A5	A4
			A4	A3
			A3	A2
			A2	A1
			A1	A0
			A0	B9
			B9	B8
			B8	B7
			B7	B6
			B6	B5
			B5	B4

MINISYMBOL TABLE

The upper right-hand portion of the screen is dedicated to a minisymbol table. There is room for 15 variables in this table. This feature is designed to let you monitor the contents of selected variables without worrying about where they physically reside. Two-byte values are displayed in high-low order (even though they're generally stored in low-high order). This symbol table is located three bytes past the beginning of the DDT code. The first three bytes are a JMP DDT ENTRY instruction. 135 locations are reserved for the minisymbol table. Each symbol in the table is in the following form:

NAME	LOCATION	BYTES to SHOW
6 characters for symbol name	symbol address 2 bytes	1 or 2 1 byte

An example of setting up a minisymbol table using the ATARI Macro Assembler (AMAC) would be :

```
ORG   DDT+3     ; This sets AMAC positon to start of symbol
                ; table
DB   'VAR1  '   ; Exactly 6 characters please!
DW   VAR1       ; Let the assembler figure out what value
                ; to put here,
DB   1          ; either  a 1 or a 2 to indicate that the
                ; variable should be shown as a single-byte
                ; or double-byte value.
```

You can also use the minisymbol table to keep an eye on standard system variables:

```
DB   'COLPF2'
DW   710
DB   1
```

You can monitor a small area of memory by setting up several dummy variables, each pointing to one or two successive bytes of memory.

The minisymbol table has other serendipitous uses. For example, you can define a program label as a symbol. The value shown will be meaningless, but the disassembly listing in the display window will be more readable:

```
DB   ':LOOP1 '
DW   :LOOP1
DB   1
```

Indeed, you can even define a symbol as "------" or some such to separate different usage areas of the symbol table. Finally, you can use the minisymbol table to help locate a portion of your code. To do this you need to set up a dummy storage location:

```
LCODE     DW   :CODE
```

You would then define the symbol variable in the table as :

```
DB   'LCODE '
DW   LCODE
DB   2
```

The value displayed will then be the address of the :CODE module.

You need not define any more symbols than you want to use. Examine some of the example programs to get a better idea of how to use the minisymbol table in various ways. Note that your definitions should be the last thing included in the shell program. This is to make sure

the symbol definitions occur after DDT, which initially sets up the table as follows :

```
ORG :SSYMT
ECHO 15
DB '        '
DW 0
DB 1
ENDM
```

BREAKPOINT TABLE

The breakpoint table is located just above the register display. There are six user-definable breakpoints and two trap breakpoints, each of which will be shown with its current setting. If a register is clear, i.e., not set, then the value shown will be 0000. If a breakpoint register is set, the value in that register will be the location of where in memory a BRK instruction has been placed. However, in the case of the TRAP breakpoints, no BRK instruction is used. These values are used in interpretive mode to create the equivalent of a break instruction.

COMMAND WINDOW

The extreme right-hand part of the bottom of the screen is devoted to the command window, the area showing the commands you type in.

TRAP

The trap breakpoints are reserved for interpretive mode. In this mode, breakpoints in memory are ignored, since DDT already has control of the system. Instead DDT checks the values in the TRAP registers. If either equals the address of the next instruction to be executed, DDT will halt the interpretive mode. This allows pseudo breakpoints to be placed in ROM locations, for instance. Then it becomes much easier and quicker to reach a certain spot in the ROM code by setting a trap, and running in interpretive mode than by single-stepping up to the desired location.

BREAKPOINTS

One of the most common debugging techniques is to use what is known as a breakpoint. Suppose you're trying to debug a program that is clobbering the system. One of the first things you can do is look at your source code and say, I wonder if it ever makes it this far. You then place a "breakpoint" or literally a BRK instruction that will call DDT. Thus, when you run your program you will find out one of two things. If your code hits the breakpoint and calls DDT, then the problem is beyond that point. However, if the program bombs and it never makes it to the breakpoint, you know the problem is prior to that point. Thus you have begun localizing the bug. Repeating this process can eventually locate where in your code the problem resides.

The breakpoint mechanism is the most common way for you to transfer control to DDT. When a program is running, executing a BRK instruction calls DDT, provided DDT has been initialized. This causes the DDT screen display to activate, and also turns on the keyboard and the function key command interpreter. The breakpoint remains set even after it has been encountered in code execution.

After a breakpoint has been encountered, and control has been transferred to DDT, there are several ways to leave DDT. The "C" command sets a breakpoint at the current location and then continue code execution. START simply continues code execution. "G" can be used to transfer control to another location.

Up to six breakpoints can be in place at any one time. The location of the breakpoints is shown in the breakpoint register display. If a breakpoint is clear (i.e., not set), it will show up as 0000. Setting a breakpoint register to a new location automatically restores an existing breakpoint, if one is already set for that register. Note also that there is an internal system breakpoint 0 used by the "C" command. If any breakpoint (including the "C" breakpoint itself) is encountered and control is transferred to DDT, then the internal "C" breakpoint is cleared.

FUNCTION KEY CONTROLS

The three ATARI Computer console function keys are used by DDT for special effects.

START - is used to continue code execution at the location indicate by the PC register. All 6502 registers are updated with th current displayed contents before control is transferred.

SELECT - is used to toggle back and forth between the DDT screen and whatever screen dynamics were active before DDT was called. An attempt has been made to allow most alternative features such as mixed Display lists, VBLANK routines, alternative character sets, display list interrupts, playfield size changes, and player-missiles.

OPTION - is used to single step the processor. This causes the disassembly filter to be turned on, but will not automatically toggle the display screen. See Single Step section for more information.

THE COMMAND INTERPRETER

The command interpreter is a code module that lets you issue keyboard commands to DDT. The command window is shown in the lower right-hand portion of the display screen. The left-hand part of this display is used for showing the register state of the machine.

Each command is a single keystroke command. However, depending upon the command, additional arguments might be required. If the key typed is not a valid DDT command, it will be ignored.

The DDT keyboard commands are :

```
E <addr>............... - Examine address addr
C ..................... - Continue, and leave breakpoint
G <addr>............... - Go to address addr
B <1-6>,<addr>......... - Breakpoint 1-6 to location addr
R <PC,A,P,X,Y,S>,<val> - Register selected is loaded with val
D <hstring>............ - Deposit hex string
↓ ..................... - Pull display window down
↑ ..................... - Push display window up
I ..................... - Interpretive mode
W ..................... - Window filter toggle
T <1-2>,<addr>......... - Trap at address
S <hstring>............ - Search for hex string
```

These commands are described in the following pages.

ENTERING A VALUE

Several of the keyboard commands require that you enter one or two values. A value entry is terminated by typing a delimiter (either a space, a comma, or a RETURN). When two values are needed, as with the Breakpoint command, a comma will be displayed after the first delimiter is typed, regardless of which delimiter was actually typed. Typing a delimiter without having entered a value will result in the entire command being ignored (exceptions--see the Breakpoint and the Trap Commands).

In the explanations that follow, these abbreviations are used :

```
<addr>           = an address value, up to 4 hexadecimal digits
                   (sorry, HEX only)
<1-6>            = either a 1,2,3,4,5 or 6
<PC,A,P,X,Y,S>   = either PC,A,P,X,Y or S
<byte>           = a single-byte value, up to 2 hexadecimal digit
<val>            = a value, which can be a byte value or an
                   address value, depending on the register chose
<hstring>        = a hex string up to 10 characters (i.e., 5 hex
                   digits)
```

The command interpreter (CI) ignores keys other than 0-9 and A-F for value inputs. To erase a character, use the DELETE key.

Each time a value is expected, the CI sets up a field size corresponding to the maximum number of hex digits that should be entered (e.g., 4 digits for an address value). When this number has been reached, no additional digits will be allowed. You can, however, delete characters, and then enter new characters. Deleting past the starting point of the value field will result in the entire command being erased.

EXAMINE E <addr>

Use the EXAMINE command to set the display window to view an area of memory. The extreme left-hand edge of the display window has a greater than sign (>) in the fourth row pointing to the current position that was entered as the address in the "E" command. Note that the "E" command does not change the state of the display window filter, nor will it affect which instruction will next be executed by a single step command.

CONTINUE C

Use the CONTINUE command to return to the code that called DDT and continue execution. It functions similarly to the operation of the START button in that execution will continue at the address indicated by the PC register. However, "C" also leaves an additional "system" breakpoint behind. Internally, this is accomplished by single stepping past the instruction, and then setting an internal, invisible breakpoint register to the location just left. Only one internal breakpoint can be maintained. If one has already been set, it will first be restored before setting the new one. This breakpoint will be cleared whenever any breakpoint (including the C breakpoint itself) is encountered during code execution.

GO G <addr>

Use the GO command to begin execution at a specific location in memory. Before control is transferred to this location, all registers are updated based upon the current contents of the displayed registers. This is true for all commands involving code execution.

BREAKPOINT B <1-6>,<addr>

Use the BREAKPOINT command to set one of the six breakpoint registers to a location. If a value other than a 1 - 6 is entered for the breakpoint register, the command will be immediately terminated. Note that two values (the breakpoint register number and the breakpoint location) are required for this command. Both fields must be terminated with a delimiter (e.g., type "B", then "1", then SPACE, then "A000", and then press RETURN). Remember, all delimiters (space, comma, and RETURN) are treated identically.

When a breakpoint is set, that location should show up in the breakpoint register display. Physically, a "0" for the BRK instruction is stored in memory at the requested location. If an EXAMINE command is issued to look at that part of memory, a "0" will be seen, even though the proper mnemonic is shown in the disassembly. If a breakpoint is set at an examined location, the mnemonic will be shown in inverse video. If a breakpoint register is already in use when a new breakpoint is requested, the instruction at the old

breakpoint is first restored.

To clear a breakpoint register and restore the source code, type any delimiter after selecting the desired breakpoint register (e.g., typing "B", then "1", then comma, and then comma will clear breakpoint 1 and restore the source code). Trying to clear a breakpoint that is not set will not harm anything. Note, however, that trying to set a breakpoint in ROM, in hardware registers, or in non-existent RAM may do some interesting things, but probably not what you wanted.

REGISTER R <PC,A,P,X,Y,S>,<val>

Use the REGISTER command to modify the contents of any of the 6502's registers. After typing "R", only a "P","A","X","Y",or "S" will be allowed. Any other character will result in the command being terminated. If an "A","X","Y", or "S" is typed, no other character other than DELETE will be allowed until a delimiter is typed. If "P" is typed, an additional "C" will be allowed to indicate the Program Counter. "P" by itself indicates the Processor Status register. "A", "P", "X", "Y", and "S" will accept only two hex digits (i.e., one byte), while "PC" will accept four digits. Note that this command requires two separate values and two separate delimiters.

WARNING! Indiscriminate use of this command, particularly with "P", "PC" and "S" can really mess things up.

DEPOSIT D <hstring>

Use the DEPOSIT command to place a string of bytes in memory. A string of hexadecimal values (up to 10 characters, 5 hex bytes) may be entered. The values entered will be placed in successive locations, starting at the current position indicated in the display window and replacing whatever was there. The input string is decoded two characters per hex byte at a time. If there is an odd character left at the end, it will be interpreted as the low order nibble of a hex value. For example, entering a string of 01AAB0 will result in three bytes (01, AA, and B0) being placed in memory. However, entering 01AAB will result in 01, AA, and 0B being deposited. Note that depositing a byte or a series of bytes will not move the display window. This must be down with the EXAMINE or the PUSH or PULL window commands.

PULL WINDOW DOWN down-arrow

Use the PULL WINDOW command to pull the display window down. Depending on the display filter in place, this will pull the window down one byte (hex filter) or by one full instruction (disassembly filter). Note that auto-repeat on the keyboard is active, so that continuing to press the down-arrow key (pressing the CTRL key isn't necessary) will continue to pull the window down.

If the SHIFT key is held down while typing the down-arrow character, the screen will be pulled down a full screen each time.

PUSH WINDOW UP up-arrow

Use the PUSH WINDOW command to push the display window up. Depending on the display filter in place, this will push the window up one byte (hex filter) or by one full instruction (disassembly filter). Again the auto-repeat on the keyboard is active, so that continuing to press the up-arrow key (pressing the CTRL key isn't necessary) will continue to push the window up.

If the SHIFT key is held down while typing the up-arrow character, the screen will be pushed up a full screen.

A problem occurs, however, when you arbitrarily examine an area of memory with the disassembly filter in. If you try to push the window up, there is not enough information to be able to tell if the preceding instruction was one, two, or three bytes long. DDT keeps track of how many bytes the window is moved each time you pull the window down. Thus, you can push the window back up if you have previously pulled it down past an instruction or group of instructions. Refer to the technical appendix for information on this feature.

INTERPRETIVE MODE I

Use the INTERPRETIVE MODE command to place the system in an automatic single step mode. After each instruction is interpreted, the screen display is updated if the DDT screen is turned on. The display window is automatically placed in the disassembly mode. Pressing the BREAK key halts the interpretive mode. It is possible to run ROM programs, such as BASIC, interpretively, but problems with the display can arise in trying to run portions of the operating system interpretively. The Trap register is used for setting up the equivalent of a breakpoint in this mode. Interpretive mode runs much faster if the user screen is selected rather than the DDT screen. This occurs because DDT does not have to update its screen if it isn't active.

WINDOW W

Use the WINDOW command to change the "filter" over the display window. "W" toggles between the filters. Three filters are available, an opaque filter with DDT operating instructions printed on it, a disassembly filter, and a hexadecimal filter.

TRAP T <1-2>,<addr>

Use the TRAP command to set one of the Trap breakpoints to a specific location. The address entered should show up in the proper Trap register. Note the trap will work only when in interpretive mode. To clear the trap, type "T", a "1" or "2" for the Trap register you want to clear and then type any two delimiters. A 0000 should show up in the register.

SEARCH S <hstring>

Use the SEARCH command to locate a specific sequence of hex characters in memory. You may enter a hex string of up to 10 characters (5 bytes). Memory will be searched from the current position indicated in the display window, up through memory, and to location C000. Since this represents memory address space that is unavailable in the system, no search

match is attempted in this area. Note that you can still look through the OS ROM by examining F111, for example, and then starting the search. Memory from F111 to FFFF will first be searched, and then 0000 to C000. If the search is successful, the display window will be repositioned. If it is unsuccessful, the command window will simply be cleared for the next command.

DDT ENTRY POINTS

There are three ways to enter DDT:

```
FLASH ENTRY
WARM ENTRY
BREAKPOINT ENTRY
```

FLASH ENTRY

This entry point is provided to allow immediate entry to DDT regardless of other circumstances. This is a single keyboard special character, and is initially set up as [CTRL] [SHIFT] [ESC] (i.e., pressing the CTRL, the SHIFT and the ESC keys at the same time). When DDT is initialized, the operating system code that looks at the keyboard is modified so that it looks for the special character first before handling normal keyboard input. If this character is found, DDT is entered immediately, through the FLASH ENTRY point.

The "C" command, or pressing START will return control to wherever the processor was when the DDT special character was typed. For more information on the Flash entry mechanism, see the Keyboard Scanner section in the Technical Details appendix.

Warning! Never use the FLASH entry twice to get to DDT without first exiting DDT. Doing so would make it impossible to return to the original calling point.

When you use the Flash entry, you will notice that the current position indicated is at a code sequence as follows :

```
PLA
TAX
PLA
TAY
PLA
RTI
```

This is a portion of the DDT code that simulates a breakpoint to enter DDT. To get to the actual machine code instruction that would next be executed, simply do six single steps.

WARM ENTRY

This entry point is the starting point for the DDT code. The first three bytes are a JMP DDT ENTRY instruction. If this location is called via a JSR instruction, then the START button exit will return control to the calling point. This allows DDT to be called at various program locations for setting up breakpoints, changing values, and so on.

Example

```
        .
        .
  -- your code --
        .
        PHA              ;this doesn't mean anything, only an example
        JSR DDT
  -- Pressing START will return here --
        .
        .
```

When you use the Warm entry, the current position will be pointing to an RTS instruction. As with the Flash entry, this is actually a portion of DDT used to implement the entry mechanism. Single step once to get to the application code that would next be executed.

BREAKPOINT ENTRY

Breakpoint entries are the most common way to enter DDT. The breakpoints first have to be set up via a FLASH or WARM entry to DDT. After they are set, DDT will be called if those specific instructions are executed. Exits from DDT breakpoints return to the code sequence where the breakpoint was located. Notice that the breakpoints will remain in place unless they are explicitly cleared. This is true even if a breakpoint has been tripped.

Recall also that if the trap register is set in interpretive mode, then attempting to execute the instruction at that address will halt the interpretive mode. Thus to move past a trap breakpoint in interpretive mode, you have to either clear the trap or single step past the instruction that was trapped and then enter interpretive mode.

HOW TO USE DDT

THE EXAMPLES

DDT contains several program examples of how to set up DDT in different ways. Turn on your computer and play with DDT as you read along.

LOADING DDT INTO COMPUTER MEMORY

1. Insert the ATARI BASIC Language Cartridge into the cartridge slot of your computer.

2. Have your computer turned OFF.

3. Turn on your disk drive

4. When the BUSY light goes out, open the disk drive door and insert the DDT diskette with the label in the lower right-hand corner nearest to you. (Use disk drive 1 if you have more than one drive.)

5. Turn on your computer and your TV set. The program will load into computer memory and display the READY prompt of ATARI BASIC.

So far everything seems normal, right? You might even want to type in a short program, such as :

```
10 FOR I=0 TO 1000
20 PRINT "I=";I
30 NEXT I
40 GOTO 10
```

Type RUN and start the program. Now then, press the CTRL key, the SHIFT key and the ESC key (all located along the left-hand edge of the keyboard) at the same time. Eh VOILA! Welcome to DUNION's DEBUGGING TOOL, better known as DDT.

There are several assembly language program "SHELLS" you should look at. This requires that you use the ATARI Program-Text Editor (MEDIT). The basic idea behind the "shell" concept is to leave the actual source code modules (DDT.MAC, DDTLST.MAC, and the source code module you're debugging) as undisturbed as possible. With a shell, you can make most necessary changes (re-orging, and so on) in the shell program and not change the other files. Each of these shells is described in the next section.

ATTACHING YOUR PROGRAM TO DDT

The assembly language program named SHELL.MAC is the general program you should use in assembling your program with DDT. A printout of this program is included in the Technical Details section of this manual. Take a look at this printout. As you can see, the SHELL program is itself a step-by-step guide to attaching DDT to your program. Let's say you have a program you normally assemble using the Macro Assembler via a source line command of:

```
D:YOURPROG.MAC S=D:SYSTEXT.MAC
```

The general procedure you would follow would be to load SHELL.MAC with MEDIT, edit it by

following the instructions in SHELL.MAC, save the file, and assemble it with a source line of:

```
D:SHELL.MAC S=D:SYSTEXT.MAC.
```

This will produce an object file called SHELL.OBJ, which in general can be renamed as an AUTORUN.SYS file that will load automatically when you turn your computer on.

Several other SHELL programs illustrate how to customize this process. Each of the SHELL programs describes how they have been customized. To see how any of these versions works, rename the desired object code file as AUTORUN.SYS and reboot the system (e.g., rename SHELL1.OBJ as AUTORUN.SYS). Unfortunately, due to space constraints, I wasn't able to leave object code modules for each of the shells. SHELL2.OBJ exists as the current AUTORUN.SYS file, and SHELL3.OBJ isn't there at all. To produce this file, you would need to assemble SHELL3.MAC.

SHELL1.MAC is a stand-alone version designed primarily to let you experiment with DDT. The variables in the minisymbol table are some of those that the operating system uses in controlling the system. This version of DDT can be helpful in understanding some of the graphic and other features of the system. You can easily examine and change the screen memory, display lists, shadow registers, and so on. You might even place a small machine-language program in memory by using the DEPOSIT command.

There are a couple of things to note about this version. First, if you use the START button exit from DDT, or the "C" command, then the DUP.SYS file will be loaded, overlaying DDT. After this happens you must reload DDT to re-enter it.

Second, since the WARMSTART mechanism is used to enter DDT, you should NOT use the FLASH entry to re-enter DDT. This will make it impossible to get to DUP.SYS via the normal exits.

SHELL2.MAC is a version that lets you examine the inner workings of a BASIC program. Notice that the variables defined in the minisymbol table are the variables BASIC uses to manage memory. One interesting thing you can do is to start a BASIC language program running, press CTRL-SHIFT-ESC to get to DDT, press SELECT to see the BASIC screen, and then press I to run the BASIC program interpretively. This effectively slows BASIC down by a factor of a hundred or so. Thus, you can let the BASIC program run until it reaches a spot you're interested in, and then press BREAK to stop the interpretive mode and return to the DDT screen. Then, use DDT to examine exactly what BASIC is doing internally.

SHELL3.MAC is a version designed for testing an assembly language subroutine. A routine on the diskette called PSEUDO.MAC is an implementation of a pseudo random number generator. Essentially, this routine will generate a pseudo random number less than or equal to a variable "upper limit". For more information on how this subroutine works, look at the source code using MEDIT. After assembling SHELL3.MAC, rename the object file, SHELL3.OBJ, as AUTORUN.SYS . Then, rebooting the system will load DDT and PSEUDO, initialize DDT, and then do a JSR DDT for initial breakpoint setting, and so on.

With this version, you should start to get an idea of the power of DDT. First off, if you're testing a subroutine dealing with numerical values (as does PSEUDO), then there is no need to set up an involved printing routine to check the output of the routine. It's very simple to place the result in a location and set up that location as an entry in the

minisymbol table.

Next, notice how the minisymbol table can be useful in several ways. A symbol can be used to monitor a routine's output (e.g., VALUE), input parameters (e.g., UPPER & DEGRAN), and even small areas of memory (e.g., RANNUM + RANNM2 = 4 contiguous bytes). However, the symbols can just as easily be defined as locations (i.e., labels) in your source code. Their value on the screen will probably be meaningless, but the disassembly listing becomes much more readable. You can even waste a variable calling it "......" to separate symbol variables from symbol labels.

To get an idea of how to use DDT, copy SHELL3.OBJ as AUTORUN.SYS, remove any cartridges and reboot your system. It should come up directly into DDT. Type "W" to toggle the screen, then press OPTION twice to single step to the start of the driver code for PSEUDO. Set a breakpoint at the location where there is a JMP LOOP instruction (you can look for this location by pulling the display window down; it should be at $4018). Now press START. The screen should flash and DDT should return with the PC set at $4018. Continue to do this. Note each time that the contents of VALUE are less than or equal to UPPER. Now experiment. Set the TRAP to $4018, then run interpretively, and so on.

SHELL4.MAC is a version designed for debugging a hybrid program (i.e., part BASIC, part assembly language). The object code here consists of the pseudo random number generator routine, the link to BASIC, and DDT. To use this version, rename SHELL4.OBJ as AUTORUN.SYS and reboot the system. When you see the READY prompt, type RUN "D:PSEUDO" , and press the RETURN key.

In the BASIC program PSEUDO, you can reset the "seed" or starting point for the pseudo random number generator. Try setting the seed to some value, and entering values for the upper limit and number of values to generate. Note the pseudo random numbers generated. Now go back and reset the seed to the same value you chose earlier. Also pick the same values you had selected for upper limit and pseudo random numbers to generate. You should get the same list of numbers. This is, of course, the power of a pseudo random number generator--the ability to generate numbers repeatedly that appear to be random.

INTERACTIONS WITH DOS

If you decide to set the origin of DDT to sit right on top of the FMS portion of DDT, you should be aware that this is exactly where DUP.SYS will load. Thus, if you try to load DUP.SYS (using the DOS command from BASIC, for example), then it will overlay DDT. No real problems will ensue from this operation, but you might run into some difficulty in trying to reload DDT from DOS. For instance, you must have created a MEM.SAV file before the operating system will let you overlay DUP.SYS. In general, if you need to use DUP.SYS , then you should ORG DDT beyond where DUP.SYS will load.

If you want to call DOS from DDT, there are several ways to do so. One simple way is to have an instruction in your code like:

```
DOSCALL        JMP (DOSVEC)       ;DOSVEC =$0A
```

Then, to call DOS, use a DDT "G" command with the address of DOSCALL.

APPENDIX - TECHNICAL DETAILS

KEYBOARD SCANNER

During DDT initialization, the system keyboard vector is redirected to a preprocessor which checks for the DDT FLASH ENTRY special character. If this character is seen, control transfers to the FLASH ENTRY point; otherwise, control passes to the normal keyboard processing routine.

When writing applications, you need to understand a couple of things about this preprocessor feature.

1. Keyboard interrupts must be enabled.

2. The character watched for is stored in an internal table and may be changed. In the source code the table location is DBCHR, which is initially set to $DC.

SINGLE STEPPING

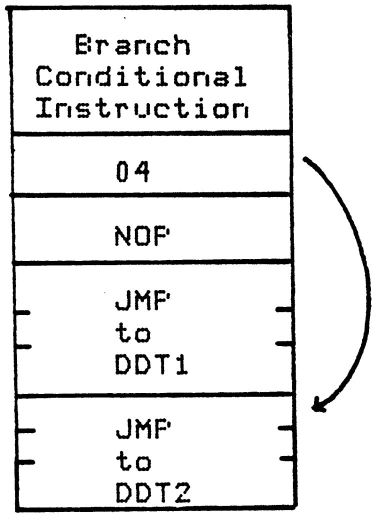
DDT is equipped with a single step mechanism for detailed examination of code execution. This is invoked by pressing the OPTION button or via the "I" command. The "I" command activates a single step automatic mode which is terminated by pressing the BREAK key.

When a single step request is issued, an examination is made of the instruction pointed to by the PC register. If it is not a "forbidden" instruction (i.e., one that could mess up DDT), it is transferred to a test bed, the 6502 registers are loaded from the register shadows, and then the instruction is executed directly. After execution, the register state is saved, the screen display is updated, and control returns to DDT.

If DDT cannot allow the instruction to be executed directly (e.g., a JMP instruction), then the instruction is simulated and the saved register state and display are properly updated before control is returned to DDT. Forbidden instructions include all branch instructions, JMP, Jump indirect, JSR, RTI, RTS and BRK.

If a breakpoint is encountered during single stepping, DDT gets the actual instruction that should be at that location before executing it. If, for some reason, the BRK instruction you are single stepping past does not correspond to one of DDT's breakpoints, an NOP will be loaded instead. This is also the case if the instruction is undefined.

The branch instructions are handled in a hybrid manner. The actual branch instruction is placed in a test bed, as shown below. Thus, after execution of the branch instruction, DDT can infer where the branch instruction with the real offset would have gone. This value is used to update the resultant address that will be placed in the PC.



DDT's USE OF SYSTEM RESOURCES

The DDT code itself occupies about 6K of RAM, and the display screen another 1K. Extreme care has been taken to ensure that DDT runs parallel to normal system functioning. In interpretive mode, for example, you should be able to use all the system's features (including the keyboard and the function keys), except for the BREAK key, which DDT reserves for itself. One underlying assumption in DDT is that your program is going to be generally operating according to the protocols established by the existing operating systems. There are six page zero locations that DDT uses when active, 2-7. The operating system will not be using these during the time DDT is active. In the event that your program uses these locations (naughty! naughty!), they are saved upon entering DDT and restored upon exit. However, if they are examined while DDT is in control, they will reflect DDT values, and not your program's values.

DDT has only two global variables, DDTI and ECODE, both of which are used in SHELL.MAC. Otherwise, all variables are local. The shell programs themselves also use global variables DDT and ICODE.

DISPLAY WINDOW MOVEMENT

DDT maintains a "pull stack" while the disassembly filter is in place. This means that each time you pull the display window down, DDT places the number of bytes that the window was pulled in a stack. Thus, when you want to push the window up, DDT checks to see if there are any values left in the pull stack. If so, you can push the window up. If not, nothing happens. The pull stack is cleared whenever DDT is entered, or when an EXAMINE command is typed. To conserve memory, four pull values (which will be a 1, 2, or 3) are packed into each byte in the stack. A total of 64 bytes are reserved for the stack. Thus you can pull the window down 256 times before the stack runs out, at which time the first values in the stack are lost and you can't back up as far. In computer terms, the stack is implemented as a circular buffer.

THINGS TO WATCH OUT FOR

To my continued dismay, a few GOTCHAs remain in DDT. In general, these occur when you are single stepping or running interpretively. If the interpreted code messes around with the display list, or with ANTIC, or CTIA/GTIA, then you might end up with a scrambled DDT screen. Usually this isn't fatal, just distracting. To restore the normal DDT screen, press the BREAK key to halt the interpretive mode, and then press SELECT twice.

Trying to do I/O from disk or any other real time activity in either interpretive mode or single step mode is probably going to produce a mysterious occurrence. You should set up breakpoints so that this type of I/O is done in real time, and then call DDT.

Be wary of using the FLASH entry point (entered by pressing CTRL-SHIFT-ESC) to re-enter DDT after it has been entered (but not exited). This will definitely confuse the system.

Some programs that you want to debug turn out to be too big to assemble along with DDT. If this occurs, AMAC will simply lock up and die. You can handle this by assembling one shell containing DDT and another containing the test program. True, you will have to do a little planning to make sure the ORG values are correct, and that the test code knows where DDT (and consequently the minisymbol table) and the initialization code are located. But this isn't really all that difficult to do once you've played around with DDT for awhile. After you have produced the two object code modules, rename the one containing DDT as AUTORUN.SYS. Then copy the other to AUTORUN.SYS with the append option. DUP.SYS will tack your test code to the end of the DDT code. Don't worry about the fact that the segments of code may be ORGed at different areas. The system binary loader will handle the segments properly. All you have to do is be sure the proper minisymbol table is loaded last, and the last segment has the proper initialization address loaded into the RUN vector.

Finally, going back and forth between DDT and DUP.SYS (if they overlay each other) seems to introduce unknown things into the system. If this happens, try pressing SYSTEM RESET first, and if this fails, simply reboot the system. I know, that is a real chicken way of dealing with the problem, but what do you want, egg in your beer?

```
*************************************
*
*
*  THIS IS THE GENERAL SHELL PROGRAM
*
*   YOU SHOULD ASSEMBLE THIS PROGRAM
*   TO ATTACH YOUR TEST PROGRAM
*   TO THE DEBUGGING SYSTEM.
*
*   REFER TO THE DDT DOCUMENTATION
*   FOR INSTRUCTIONS ON CUSTOMIZING
*   THIS PROGRAM FOR YOUR PARTICULAR
*   NEEDS.
*
*
*
*************************************
*
*
*
*             STEP 1
*
*   FIRST YOU HAVE TO DECIDE WHERE
*   DDT AND YOUR CODE WILL RESIDE.
*
*   ONE CHOICE IS TO LET DDT SIT
*   RIGHT ON TOP OF DOS, AND IN A
*   SENSE, BE AN EXTENSION OF IT.
*
*   IN THIS CASE THE ORG STATEMENT
*   SETS DDT TO BEGIN RIGHT WHERE
*   DOS STOPS. NOTE THIS IS THE
*   STANDARD 2 DISK DRIVE DOS
*   CONFIGURATION.
*
*   IF YOU HAVE SPECIAL CONDITIONS
*   (FEWER DISK DRIVE BUFFERS, THE
*   850 ON, ... ) THEN CHANGE THE
*   ORG TO SUIT YOUR TASTE.
*
        ORG $1CFC
*
*************************************
*
*       STEP 2
*
*   NOW YOU HAVE TO MAKE SURE THE
*   DDT CODE IS ASSEMBLED.
*   HERE, IT IS ASSUMED THAT ALL
*   THE NECESSARY FILES ARE LOCATED
*   ON DRIVE 1.
*
*   YOU CAN CHANGE THE FILE
*   DESIGNATORS HOWEVER, TO FIT
*   YOUR DEVELOPMENT SYSTEM.
*
        PROC
DDT     =       *
        INCLUDE D:DDT.MAC
*
*************************************
```

```
**************************************
*
*       STEP 3
*
*   THE NEXT FILE IS THE DISPLAY LIST
*   AND SCREEN AREA FOR DDT.
*
*   THIS CODE TAKES UP JUST UNDER 1 K
*   OF MEMORY SPACE, AND HAS SOME
*   BOUNDARY CROSSING RESTRICTIONS.
*
*   NOTE THAT THE FOLLOWING ORG
*   STATEMENT ASSURES THAT THE DISPLAY
*   LIST DOES NOT CROSS A 1K BOUNDARY
*   AND THAT THE SCREEN MEMORY DOES
*   NOT CROSS A 4K BOUNDARY.
*
*   IF YOU WANT TO MOVE THE SCREEN
*   FOR ANY REASON, MODIFY THIS
*   STATEMENT.
*
*   NOTE ALSO THAT THE ICODE LABEL
*   IS USED TO DEFINE A SPOT TO
*   STORE INITIALIZATION CODE.
*   9 BYTES ARE SAVED FOR THIS
*
        IF [[[[[[HIGH *]/4]+1]*1024]-*]<33] OR [[[[[HIGH *]/8]+1]*1024]-
        ORG [[HIGH *]/4]+1]*1024
        ENDIF
*
        INCLUDE D:DDTLST.MAC
        EPROC
ICODE   = *
        ORG *+9
*
**************************************
*
*       STEP 4
*
*   THE DDT INITIALIZATION CODE SETS
*   UP A ROUTINE THAT MODIFIES THE
*   MEMLO POINTER WHENEVER THE RESET
*   BUTTON IS PRESSED.
*
*   NORMALLY THIS IS USED TO "HIDE"
*   THE DDT CODE, AND MAKE THE FREE
*   MEMORY AREA START JUST AFTER DDT
*
*   TO MODIFY THIS SET UP YOU WILL
*   HAVE TO DEFINE AN ECODE VALUE TO
*   BE PLACED IN THE MEMLO POINTER.
*
*   ONE SUGGESTION WOULD BE TO SIMPLY
*   PUT THE NORMAL VALUE THAT WOULD
*   BE THERE ANYWAY.
*
*   FOR INSTANCE IN THE STANDARD DOS
*   CONFIGURATION, YOU MIGHT PUT
*   ECODE   =   $1CFC
*
```

```
****************************************
*
*       STEP 5
*
*   NOW YOU HAVE TO ATTACH YOUR OWN
*   CODE. A COUPLE OF THINGS SHOULD
*   BE NOTED.
*
*   1. YOU SHOULD REMOVE ANY ORG
*      STATEMENTS FROM YOUR CODE
*      AND PLACE THEM HERE.
*      WITH NO NEW ORG STATEMENT,
*      YOUR CODE WILL FOLLOW THE
*      DDT CODE.  CURRENTLY THAT
*      MEANS YOUR CODE WOULD START
*      AROUND $3715
*
*   2. REMOVE ANY END STATEMENT FROM
*      YOUR PROGRAM. IF NOT, IT WILL
*      DEFINITELY SCREW THINGS UP.
*
        ORG     YOURORG
        INCLUDE D:YOURPROG
*
****************************************
*
*       STEP 6
*
*   IF YOU WANT TO DEFINE A MINI
*   SYMBOL TABLE, THIS IS THE SPOT.
*   THE ORG STATEMENT SHOULD SET THE
*   ORG TO WHEREEVER DDT IS +3
*
*   RECALL THAT EACH SYMBOL NEEDS TO
*   BE DEFINED LIKE :
*
*      DB    'SYMBOL' ;6 CHARACTERS
*      DW    SYMBOL   ;SYMBOL LOCATION
*      DB    1        ;A 1 OR 2
*
        ORG DDT+3
        DB       '      '
        DW       0
        DB       1
*
****************************************
```

```
*************************************
*
*       STEP 7
*    NOW YOU HAVE TO TELL THE SYSTEM
*    WHERE TO GO TO RUN THE CODE
*
*    THE STRUCTURE WE HAVE HERE WILL
*    INITIALIZE DDT, CALL DDT TO
*    ALLOW YOU TO SET UP INITIAL
*    BREAKPOINTS, AND THEN JUMP TO
*    THE START OF YOUR CODE.
*
        ORG  ICODE
        JSR DDTI    ;INITIALIZE DDT
        JSR DDT     ;ENTER DDT
        JMP YOURORG ;AND RUN YOUR CODE
*
*
        END      ICODE
*
*************************************
```

Limited Warranty on Media and Hardware Accessories. Atari, Inc. ("Atari") warrants to the original consumer purchaser that the media on which APX Computer Programs are recorded and any hardware accessories sold by APX shall be free from defects in material or workmanship for a period of thirty (30) days from the date of purchase. If you discover such a defect within the 30-day period, call APX for a return authorization number, and then return the product to APX along with proof of purchase date. We will repair or replace the product at our option. If you ship an APX product for in-warranty service, we suggest you package it securely with the problem indicated in writing and insure it for value, as Atari assumes no liability for loss or damage incurred during shipment.

This warranty shall not apply if the APX product has been damaged by accident, unreasonable use, use with any non-ATARI products, unauthorized service, or by other causes unrelated to defective materials or workmanship.

Any applicable implied warranties, including warranties of merchantability and fitness for a particular purpose, are also limited to thirty (30) days from the date of purchase. Consequential or incidental damages resulting from a breach of any applicable express or implied warranties are hereby excluded.

The provisions of the foregoing warranty are valid in the U.S. only. This warranty gives you specific legal rights and you may also have other rights which vary from state to state. Some states do not allow limitations on how long an implied warranty lasts, and/or do not allow the exclusion of incidental or consequential damages, so the above limitations and exclusions may not apply to you.

Disclaimer of Warranty on APX Computer Programs. Most APX Computer Programs have been written by people not employed by Atari. The programs we select for APX offer something of value that we want to make available to ATARI Home Computer owners. In order to economically offer these programs to the widest number of people, APX Computer Programs are not rigorously tested by Atari and are sold on an "as is" basis without warranty of any kind. Any statements concerning the capabilities or utility of APX Computer Programs are not to be construed as express or implied warranties.

Atari shall have no liability or responsibility to the original consumer purchaser or any other person or entity with respect to any claim, loss, liability, or damage caused or alleged to be caused directly or indirectly by APX Computer Programs. This disclaimer includes, but is not limited to, any interruption of services, loss of business or anticipatory profits, and/or incidental or consequential damages resulting from the purchase, use, or operation of APX Computer Programs.

Some states do not allow the limitation or exclusion of implied warranties or of incidental or consequential damages, so the above limitations or exclusions concerning APX Computer Programs may not apply to you.

For the complete list of current APX programs, ask your ATARI retailer for the APX Product Catalog

ATARI®
PROGRAM
EXCHANGE
P.O. Box 3705
Santa Clara, CA 95055

Review Form

We're interested in your experiences with APX programs and documentation, both favorable and unfavorable. Many of our authors are eager to improve their programs if they know what you want. And, of course, we want to know about any bugs that slipped by us, so that the author can fix them. We also want to know whether our instructions are meeting your needs. You are our best source for suggesting improvements! Please help us by taking a moment to fill in this review sheet. Fold the sheet in thirds and seal it so that the address on the bottom of the back becomes the envelope front. Thank you for helping us!

1. Name and APX number of program.

____

____

2. If you have problems using the program, please describe them here.

____

____

____

3. What do you especially like about this program?

____

____

____

4. What do you think the program's weaknesses are?

____

____

____

5. How can the catalog description be more accurate or comprehensive?

____

____

6. On a scale of 1 to 10, 1 being "poor" and 10 being "excellent", please rate the following aspects of this program:

_______ Easy to use
_______ User-oriented (e.g., menus, prompts, clear language)
_______ Enjoyable
_______ Self-instructive
_______ Useful (non-game programs)
_______ Imaginative graphics and sound

7. Describe any technical errors you found in the user instructions (please give page numbers).

8. What did you especially like about the user instructions?

9. What revisions or additions would improve these instructions?

10. On a scale of 1 to 10, 1 representing "poor" and 10 representing "excellent", how would you rate the user instructions and why?

11. Other comments about the program or user instructions:

From

STAMP

ATARI Program Exchange
P.O. Box 3705
Santa Clara, CA 95055

[seal here]